Proceedings of The National Conference
On Undergraduate Research (NCUR) 2016
University of North Carolina Asheville
Asheville, North Carolina
April 7-9, 2016

Exploring the Oriented Graceful Labeling Conjecture on Lobster Trees

Timothy Nosco, Lisa Jones, Jakub Smola, Jessie Lass
Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, New York 10997 USA

Faculty Advisors: Dr. Jocelyn Bell, Dr. William Pulleyblank, Dr. Suzanne J. Matthews and Dr. Christopher Okasaki

Abstract

Introduced in 1967 by Rosa and further explored by Ringel and Kotzig, the famously unsolved graceful tree labeling conjecture (GTL) is a labeling problem in graph theory which proposes that for any tree with m+1 vertices, each vertex can be assigned a distinct label from 0 to m such that if we assign to each edge the absolute value of the difference of its adjacent vertices, then every edge is assigned a distinct value between 1 and m. This conjecture has been proven to hold for certain families of trees, but no proof for the general conjecture has been found. In 2015, Bell et al. presented a stricter version of GTL known as the oriented graceful tree labeling conjecture (OTL), imposing the additional constraint that for every vertex v in a tree T, the labels of vertices adjacent to v are either all strictly less than the label of v, or all strictly greater than the label of v. This paper addresses the mathematical community's desire for a computational toolset to assist in the exploration of these graph labeling conjectures, particularly on lobster tree topologies. Using Python, parallel computing, and Microsoft's Z3 SMT solver, we are able to verify the conjecture for lobster trees up to 19 vertices. We plan to continue developing this toolset and eventually make it available to the public as an open-source project.

Keywords: graceful labeling, parallel computing, SMT solvers

1. Introduction

Popularized in 1963 by Ringel and Kotzig [1], the famously unsolved graceful tree labeling conjecture is a labeling problem in graph theory which proposes that for any tree with m+1 vertices, each vertex can be assigned a distinct label from 0 to m such that if each edge label represents the absolute value of the difference of its adjacent vertices, then every edge is assigned a distinct value between 1 and m. This conjecture has been proven to hold for certain families of trees, but no proof for the general conjecture has been found.

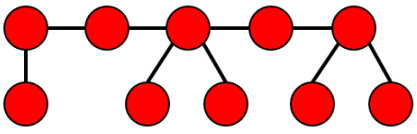

Figure 1. Example caterpillar tree with 10 vertices.

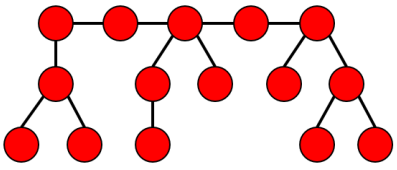

Figure 2. Example lobster tree with 15 vertices.

Rosa [2] proved in 1967 that all caterpillar trees admit a graceful labeling. Caterpillar trees consist of a single path, to which any number of leaf nodes can be connected (Figure 1). In this paper, we focus on lobster trees. In a lobster

tree, the removal of all the leaf nodes causes the tree to become a caterpillar tree (Figure 2). Note that removing the leaves from the bottom level of Figure 2 yields the caterpillar tree in Figure 1.

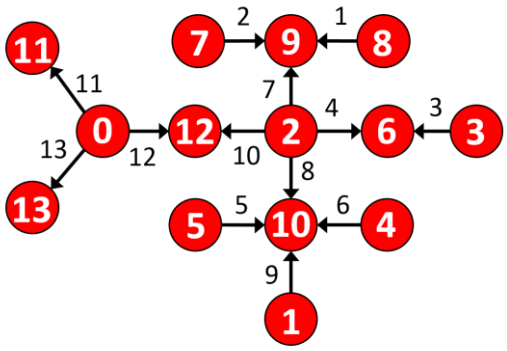

Figure 3. A gracefully labeled oriented tree with 14 vertices.

In 2008, Morgan [3] proved that all lobsters with perfect matchings are graceful. Many mathematicians believe that this presents strong evidence that an inductive proof that all lobsters can be gracefully labeled exists. In 2015, Bell et al. [4] presented a stricter version of the graceful tree labeling conjecture known as the *oriented graceful tree labeling conjecture*, imposing the additional constraint that for every vertex *v* in a tree *T*, the labels of vertices adjacent to *v* are either all strictly less than the label of *v*, or all strictly greater than the label of *v*. Figure 3 shows an oriented tree with 14 vertices and a valid graceful labeling. Notice that all vertices are uniquely labeled in white and all edges are uniquely labeled in black.. Furthermore, each edge's label is the absolute difference of the labels of its two vertices, and each vertex satisfies the oriented constraint.

This paper describes the first large-scale computational exploration of the oriented graceful tree labeling conjecture. It presents an algorithm that hybridizes an open-source SMT solver [5] with the Message Passing Interface library [6] to simultaneously prove the conjecture on multiple lobster trees. The paper's contributions also include a novel enumeration algorithm for lobster trees, as well as random tree generation capabilities that allow the user to test trees of a particular family or order. Using this framework, we were able to prove the conjecture for lobster trees up to 19 nodes. We plan to open-source our software suite, and believe that it provides a useful framework for mathematicians to rapidly prototype and assess graph labeling algorithms.

The rest of the paper is organized as follows. Section 2 of this paper outlines the core algorithms of our approach. Section 3 describes benchmarking results. Section 4 concludes the paper.

2. Methodology

The goal of this paper is to prove that the oriented graceful labeling conjecture holds for all lobster trees with *V* vertices (order *V*). Accomplishing this requires the completion of three subtasks. The first is a lobster generation algorithm, which generates all lobster trees of order *V*. The second is an SMT solver with programmed constraints that determines if an individual tree has a valid labeling. Lastly, a parallel algorithm that labels multiple trees simultaneously. The subsections below discuss each of these subcomponents in detail.

2.1. Lobster Generation

Before testing whether all lobsters of a particular order admit a graceful labeling, the program must generate those lobsters. This section discusses the necessary representation of lobster trees, the problem of isomorphisms, and finally the primary cases in the generation algorithm.

A lobster tree is a path with any number of subtrees of height 1 or 2 attached to each vertex in the path. In this paper, trees are represented as a list of tuples of integers. The outer list represents the path. The inner tuples represent the subtrees attached to the corresponding vertex in the path, where each subtree is represented by its size. For example, the lobster in Figure 1b could be represented as

[(3), ( ), (2,1), ( ), (1,3)]

It is very easy to write a naïve recursive algorithm to generate lobsters in this representation. The problem is that this naïve algorithm will generate many lobsters that are isomorphic to each other, which is not ideal. There are three sources of ambiguity in this representation that would allow isomorphisms to happen if we are not careful.

Ambiguity #1: Changing the order of the subtrees attached to a particular node does not result in a different tree. For example, these four lobsters are isomorphic:

[(3), ( ), (2,1), ( ), (1,3)]
[(3), ( ), (2,1), ( ), (3,1)]
[(3), ( ), (1,2), ( ), (1,3)]
[(3), ( ), (1,2), ( ), (3,1)]

To prevent this ambiguity, the algorithm sorts every inner tuple from largest to smallest, as shown in the bolded lobster. Note that, especially when sorted in this way, the inner tuples form *integer partitions*. The integer partitions of a number P are the (sorted) tuples of positive integers that sum to P. For example, there are 5 partitions of P=4: (4), (3,1), (2,2), (2,1,1), and (1,1,1,1). Standard algorithms exist for creating these partitions [7].

Ambiguity #2: Presenting a path from left-to-right or from right-to-left does not result in a different tree. For example, these two lobsters are isomorphic:

[(3), ( ), (2,1), ( ), (3,1)]
[(3,1), ( ), (2,1), ( ), (3)]

To prevent this ambiguity, the algorithm ensures that, if descendents of the two end vertices of the path are not isomorphic, then the outer list is oriented so that the "larger" end vertex comes first, as shown in the bolded lobster. Define larger by comparing the two inner tuples as follows:

- If the sums of the two inner tuples are different, then the tuple with the greater sum is larger. So (3,1) is larger than (3), and (5) is larger than (2,1,1).
- If the sums are equal, but the two inner tuples are different, then they are compared lexicographically by finding the first position at which they differ. The one with the greater integer in that position is larger. So (3,1) is larger than (2,2) and (4,2) is larger than (4,1,1).
- Otherwise, the two inner tuples are identical and neither is larger.

Consider these isomorphic not-quite palindromes:

[(3,1), ( ), (2,1), (4), ( ), (3,1)]
[(3,1), ( ), (4), (2,1), ( ), (3,1)]

In such situations, the algorithm works from both ends toward the middle until finding the first mismatch, in this case between (2,1) and (4), and insists that the list is oriented so that the larger of the mismatched tuples comes before the smaller, as shown in the bolded lobster.

Notice that a true palindrome is identical in both orientations and so does not need a special rule to prevent isomorphisms.

Ambiguity #3: For a given lobster, different sets of vertices can form the path. For example, consider a lobster that is a simple path of 6 vertices. It could be represented as

[( ), ( ), ( ), ( ), ( ), ( )]
[(1), ( ), ( ), ( ), ( )]
[(2), ( ), ( ), ( )]
. . .
[(2), (2)]

Essentially, for vertices at or adjacent to an end of the path, there can be ambiguity about whether that vertex should be considered to be on the path, or a child of its neighbor.

To disambiguate, the algorithm enforces that both end vertices of the path must have a subtree of size 2 or greater, as shown in the bolded lobster. With this rule, the choice of path for a lobster is exactly and unambiguously those vertices that remain after removing all the leaves (vertices of degree 1), and then removing all the new leaves. This paper refers to this unambiguous path as the *spine*. The length of the spine is 4 less than the length of the longest path in the lobster. Note that degenerate lobsters—those whose longest path is length 4 or less—do not have a spine, and can easily be handled on an ad hoc basis.

The lobster generation algorithm proceeds as follows. First, produce degenerate lobsters. For simplicity, the following description is phrased in terms of making choices as if the algorithm were creating a single arbitrary lobster. The real algorithm recursively makes all of those choices in all possible ways.

Let V represent the total number of desired vertices. First, choose a spine length S between 1 and V-4. (The -4 ensures the end vertices on the spine both have at least two descendants.) Create the outer list of length S.
Create left and right markers in the first and last positions of the list, respectively. Initialize R, the number of remaining vertices, to V-S. Also initialize a flag variable PALINDROME to true.

Repeat the following steps until done:

- If the left marker and right marker are equal, there is only one position remaining. Choose an integer partition of R and place it in the spine list at that position. Return the spine list as the next lobster.
    - If this is the first pass, require the integer partition to contain at least 2 entries of 2 or greater.
- If the left and right markers are one apart, there are only two positions remaining. Choose a number k_1 between 0 and R, and let $k_2 = R-k_1$. Choose an integer partition of k_1 and place it in the position of the left marker. Choose an integer partition of k_2 and place it in the position of the right marker. Return the spine list as the next lobster.
    - If PALINDROME is true, require the left partition to be at least as large as the right partition.
    - If this is the first pass, require both partitions to contain at least 1 entry of 2 or greater.

- Otherwise, choose a number k_1 between 0 and R, and a number k_2 between 0 and R-k_1. Choose an integer partition of k_1 and place it in the position of the left marker. Choose an integer partition of k_2 and place it in the position of the right marker. Subtract k_1 and k_2 from R. Move the left and right markers one position toward the middle.

    - If PALINDROME is true, require the left partition to be at least as large as the right partition. If the left partition is larger, set PALINDROME to false.
    - If this is the first pass, make sure that both partitions contain at least 1 entry of 2 or greater.

2.2. Satisfiability Modulo Theories

As the order of lobster trees increases, so does the complexity of labeling the trees. Therefore, in the absence of an inductive proof that all lobsters admit a graceful labeling—and, consequently, an iterative algorithm for gracefully labeling lobsters—an approach utilizing heuristics to efficiently search for a label is required. However, the approach must also ensure that all possible labels were tested if necessary. This is exactly the capability offered by constraint programming solvers such as Satisfiability Modulo Theories (SMT) solvers.

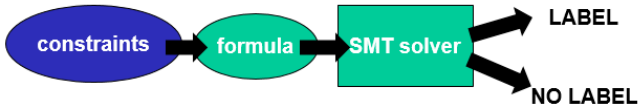

Figure 7. Overview of graceful labeling with SMT Solver

An SMT solver takes a formula in first-order logic and determines whether that formula can be satisfied. For graceful labeling, each given tree is translated into a formula that models the constraints for the (oriented) graceful labeling of that tree. The formula is then sent to the SMT solver, which eventually determines that yes, the tree can be labeled, or that no labeling exists. Figure 7 gives an overview of this process.

This paper uses Z3 [5], Microsoft's open-source SMT solver. In addition to being open-source, Z3 is capable of running on multiple threads, allowing the leverage of a shared memory architecture. When sending the formula for a particular tree to Z3, its multicore functionality enables it to search the space of valid labelings for that tree concurrently. If multiple trees are assigned to the solver, the trees are assessed sequentially.

The use of a constraint solver also increases the applicability of the work to solving other types of problems. It is for this reason that a constraint solver was chosen rather than a custom algorithm for graceful labeling. To use this framework for other types of problems, it is sufficient to reprogram the SMT solver with new constraints.

2.3. Parallel Algorithm

As the number of vertices (order) in the lobster tree increases, the number of total lobster trees (lobster space) grows exponentially. In addition, the complexity of labeling a tree increases with the number of vertices. Lobster space cannot be searched heuristically; proving the conjecture for a particular order V requires showing that all lobster trees with V vertices have a valid labeling. Exhaustively labeling each tree in lobster space quickly becomes intractable, even with the use of a multicore constraint solver such as Z3. To speed up labeling multiple trees, a wrapper is created to interface Z3 with the Message Passing Interface (MPI). This enables the division of the lobsters in that space evenly between multiple nodes. This parallel approach enables us to exhaustively test the space of possible lobster trees with larger number of vertices.

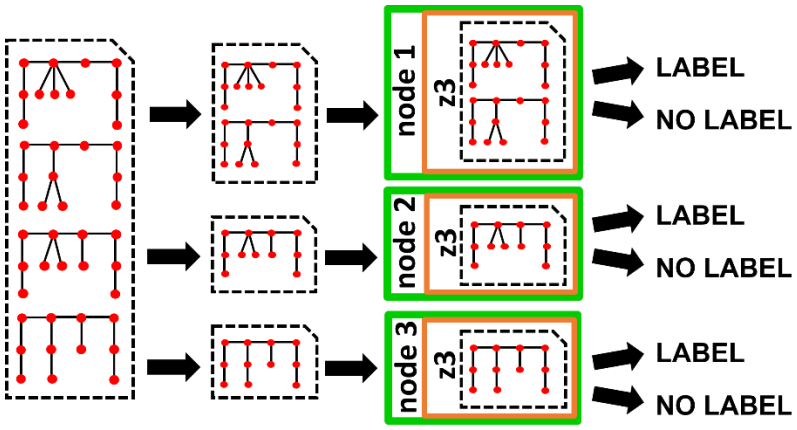

Figure 8. Overview Parallel Approach.

Figure 8 gives an overview of the parallel approach. The program first generates all lobster trees with V vertices on the master node and outputs it to a single file. This is a serial approach; experimental benchmarks (not shown) indicate that the lobster generation process is very fast, and that the bulk of computation time lies with the labeling process. Next, the program splits the file into n equally-sized chunks, where n is the number of worker nodes.Note that files are written to a distributed file system that is shared between all the nodes. Each worker node then runs Z3 on its assigned chunk of trees. In the example above, there are n=3 worker nodes. The execution of Z3 is highlighted in orange (view electronically). Each node loops sequentially through the trees assigned to it, attempting to label each tree. If a valid labeling is found for every tree assigned to a worker, the worker node sends a message to the master node indicating that it has successfully completed labeling all trees in its assigned chunk.
If a label is not found for a particular tree, the tree is stored, and the node returns that it has failed to label a tree. At this point, the MPI process is terminated, since the program has identified a tree that cannot be labeled. This case has never happened in our testing, and would in fact disprove the (oriented) graceful labeling conjecture!

3. Experimental Design and Results

Benchmarks were performed on Lightning [8], a Cray XC30 high performance computing cluster hosted out of the Air Force Research Lab D.O.D. Supercomputing Research Center (AFRL DSRC). The Lightning cluster consists of 2,370 compute nodes, each with 24 Intel Xeon E5-2697v2 processors at 2.7Ghz, 64 GB of RAM and over 1TB of available user hard disk space. The cluster uses the Lustre distributed file system. Portable Batch System (PBS) [9] is employed for job scheduling. Access was granted to the standard queue, a relatively low-priority queue for the system. This limited our ability to schedule large numbers of nodes for jobs. We ended up using approximately 150,000 compute hours over the course of our project.

Table 1. Run time (seconds) corresponding to scalability test.

	Number of Compute Nodes				
Order (V)	**1**	**16**	**32**	**64**	**128**
12	106.70	12.82	10.85	11.52	12.74
13	436.76	52.16	29.84	26.31	23.58
14	2803.83	274.70	218.29	137.19	124.28
15	14129.50	1566.28	888.59	725.81	600.68

3.1. Scalability Analysis

First, we study the scalability of our approach as the number of compute nodes is increased. Table 1 shows the run-time of the framework for labeling all possible lobster trees with 12 to 15 vertices, and varying the number of compute nodes from 1 to 128. Note that in the case of a single compute node, Z3 is still running in multithreaded form; it is however labeling each tree sequentially. For 1 compute node, the run time increases from 106.70 seconds to 14,129 seconds as we vary the number of vertices in each tree from 12 to 15. In other words, it takes Z3 132 times longer to label every tree with 15 vertices versus every tree with 12 vertices, a combined effect of there being more trees of the larger size (6739 vs 532) and each individual larger tree taking longer to label. Increasing the number of nodes has a clear effect on run time. When run on 128 nodes, the program is able to exhaustively determine labels for all lobster trees of order 15 in 600.68 seconds, approximately 10 minutes.

Figure 9 shows the speedup of the framework compared to single-node Z3 execution. For lobster trees with 12 vertices, the use of 32 nodes proved to be an inefficient use of resources. In this case, the process of serially generating all the lobster trees and splitting the files consumed more time than the actual time needed to label the trees on 128 nodes. As the lobster order increases however, the associated tree space increases exponentially. For larger orders (13-15) the use of 32 and 64 nodes greatly improved the run-time of our labeling approach. For example, for 1obster space of order 14, the program achieved a speedup of 12.8 on 32 nodes and a speedup of 20.43 on 64 nodes. For lobster space of order 15, the program achieved speedups of 15.9 and 19.4 on 32 and 64 nodes, respectively. Note that using 128 nodes started to demonstrate diminishing returns. For lobster space of order 14, the speedup is 21.98, which increases to 23.52 for lobster space of order 15.

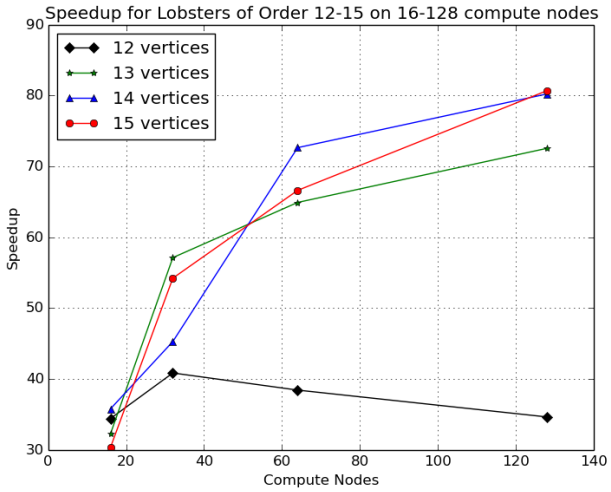

Figure 9. Speed up of our scalability test.

3.2. Race to the greatest *V*

We next focused on trying to prove the ordered graceful labeling conjecture for larger lobster spaces, with the goal of trying to prove the conjecture for the largest order possible. To date, we have successfully proven the conjecture for lobsters of order 19. Based on the results of the scalability analysis, the next set of experiments were run on 32 nodes. This decision was made because it yielded fairly good speedup and resulted in quicker job scheduling on the cluster. We referred to this process as the "race to the greatest *V*".

Table 2 enumerates our experimental results. We were able to prove the conjecture for lobster spaces up to 18 with 32 nodes. There are 89,779 lobsters of order 18. While the space of trees was equally distributed between compute nodes, certain trees proved more difficult to label than others. For lobster trees of order 18, it took each node roughly 4.5 hours to label its share of trees. However, it took one node 24 hours, suggesting that a subset of trees in that file were taking an especially long time to label. This became more evident when we increased the number of vertices to 19. For lobster trees of order 19, the job timed out on 32 compute nodes. The remaining trees were rescheduled across 64 compute nodes; all but one node were able to label their assigned trees in 10 hours. The last node timed out after 40 hours. This process was repeated until a single tree that took Z3 more than 24 hours to label was identified.

Table 2. Race to greatest *V*.

Order (*V*)	Number of Lobsters (T)	Compute Nodes (*n*)	Avg Time Per Node	Time For Slowest Node
16	15,955	32	30 minutes	1.5 hours
17	37,776	32	1 hour	4 hours
18	89,779	32	4.5 hours	24 hours
19	213,381	64	10 hours	>40 hours

We tried to label this same tree using a different type of solver called CPLEX [10]. Unlike Z3 and SMT solvers that use constraint programming, CPLEX uses an integer programming paradigm to solve optimization problems. In this case, it took less than a second to label the tree with CPLEX. We caution the reader that this does not necessarily indicate that CPLEX is "better" than Z3 at producing graceful labelings. Rather, we conjecture that the heuristics that CPLEX uses are more suited to features of the representation and the topology of this tree. Likewise, Z3 may be a more suitable solver than CPLEX for other lobster topologies.

4. Conclusions and Future Work

Using our framework, we were able to prove the oriented graceful conjecture for lobsters up to order 19. We plan to continue running our experiments to see how high an order we can exhaustively search. This process is limited by the availability of HPC resources and the amount of time it takes to narrow down any job time-outs to a single tree.

Our analysis presents several fruitful directions for future development. Although the labeling algorithm does produce all lobsters of a given order without isomorphism, it accomplishes this iteratively— that is, it does not leverage the fact that a bijection exists between $\mathbb{N}$ and lobster trees. Because of this, the algorithm cannot generate a particular lobster without first producing all the lobsters that lexicographically precede it.

Improving the lobster generation algorithm to produce lobsters independently and by bijection is the first step toward improving the load balancing of the framework. As previously noted, Z3 has great difficulty labeling certain types of trees. Although we lack sufficient data points to specifically characterize these trees, we expect that we will find more examples as we attempt to label trees with higher order. Analyzing these trees should indicate a heuristic to predict whether Z3 will take a long time to label a particular tree. The program could then preemptively distribute these hard trees equitably among the compute nodes in order to reduce the total time required to determine if a valid labeling exists.

Another interesting investigation would be to compare the performance of different solvers (e.g. CPLEX [10], MiniSAT [11]) in generating oriented graceful labels for these hard trees. Since Z3 only utilizes 8 cores per compute node, it would be possible to spawn instances of other solvers on the same node; if Z3 did not find a label in a specified amount of time, the process could pass the hard tree to another solver. Alternatively, we may be able to develop heuristics that predict which solver is best suited to trees that have particular topological features.

While our work lends credence to the claim that all lobsters admit an oriented graceful labeling, we stress that our process has much broader applicability. Our software is modular to the extent that it is trivial to substitute any set of graph labeling constraints in place of orientation and gracefulness; by importing new data structures, we can utilize the combination of SMT solvers and high performance computing to attack any problem that can be formulated in terms of constraint programming. We believe that our software will be useful to mathematicians as they explore topological problems in the future.

5. Acknowledgements

We are sincerely grateful to the High Performance Computing Modernization Program (HPCMP) at West Point, especially Dr. Claire Verhulst for facilitating access to the Lightning cluster. The opinons expressed in the paper are those solely of the authors and do not necessarily reflect those of the U.S. Military Academy, U.S. Army, or the Department of Defense.

6. References

[1] Ringel, Gerhard. "Problem 25." Theory of Graphs and its Applications, *Proc. Symposium Smolenice*. Vol. 1263. 1963.

[2] Rosa, Alexander. "On certain valuations of the vertices of a graph." *Theory of Graphs (Internat. Symposium)*, Rome. 1966.

[3] Morgan, David. "All lobsters with perfect matchings are graceful." *Electronic Notes in Discrete Mathematics* 11 (2002): 503-508.

[4] Bell, Jocelyn, et. al. Oriented Graceful Labeling of Trees. *Preprint*. 18 MAY 2016.

[5] De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008. 337-340.

[6] Gropp, William, et al. "A high-performance, portable implementation of the MPI message passing interface standard." *Parallel computing* 22.6 (1996): 789-828.

[7] Skiena, S. "Partitions." §2.1 in *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Reading, MA: Addison-Wesley, pp. 51-59, 1990.

[8] Staff. "AFRL Fires Up 1.28 Petaflop 'Lightning' Supercomputer". *Inside HPC*. September 24, 2014. Accessed from: http://insidehpc.com/2014/09/afrl-unveils-one-worlds-fastest-computers/

[9] Henderson, Robert L. "Job scheduling under the portable batch system." *Job scheduling strategies for parallel processing*. Springer Berlin Heidelberg, 1995.

[10] CPLEX, IBM ILOG. "V12. 1: Users manual for CPLEX." *International Business Machines Corporation* 46.53 (2009): 157.

[11] Sorensson, Niklas, and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization." *SAT* 2005 (2005): 53.